Empirical Hardness of Finding Optimal Bayesian Network Structures: Algorithm Selection and Runtime Prediction

Malone, Brandon

2018-01

Malone , B , Kangas , K , Järvisalo , M , Koivisto , M & Myllymäki , P 2018 , ' Empirical Hardness of Finding Optimal Bayesian Network Structures: Algorithm Selection and Runtime Prediction ' , Machine Learning , vol. 107 , no. 1 , pp. 247-283 . https://doi.org/10.1007/s10994-017-5680-2 , https://doi.org/10.1007/s10994-017-5680-2

http://hdl.handle.net/10138/309050
https://doi.org/10.1007/s10994-017-5680-2

acceptedVersion

Machine Learning manuscript No.
(will be inserted by the editor)

Empirical Hardness of Finding Optimal Bayesian Network Structures

Algorithm Selection and Runtime Prediction

Brandon Malone · Kustaa Kangas · Matti Järvisalo · Mikko Koivisto · Petri Myllymäki

Received: date / Accepted: date

Abstract Various algorithms have been proposed for finding a Bayesian network structure that is guaranteed to maximize a given scoring function. As the state-of-the-art algorithms rely on adaptive search strategies, such as branch-and-bound and integer linear programming techniques, the time requirements of the algorithms are not well characterized by simple functions of the instance size. Furthermore, no single algorithm dominates the others in speed. Given a problem instance, it is thus *a priori* unclear which algorithm will perform best and how fast it will solve the instance.

We show that for a given algorithm the hardness of a problem instance can be efficiently predicted based on a collection of non-trivial features which go beyond the basic parameters of instance size. Specifically, we train and test a statistical model on empirical data, based on the largest evaluation of state-of-the-art exact algorithms to date. We demonstrate that we can predict the running times to a reasonable degree of accuracy, and effectively select algorithms that perform well in terms of running times on a particular instance. Moreover, we also show how the results can be utilized in building an algorithm portfolio that combines several individual algorithms in an almost optimal manner.

This work is supported by Academy of Finland, grants #125637, #251170 (COIN Centre of Excellence in Computational Inference Research), #255675, #276412, and #284591; Finnish Funding Agency for Technology and Innovation (project D2I); and Research Funds of the University of Helsinki.

Brandon Malone
Section of Bioinformatics and Systems Cardiology, Department of Internal Medicine III and Klaus Tschira Institute for Integrative Computational Cardiology, University of Heidelberg, Germany, DZHK (German Centre for Cardiovascular Research), Partner site Heidelberg/Mannheim, Germany E-mail: brandon.malone@uni-heidelberg.de

Kustaa Kangas · Matti Järvisalo · Mikko Koivisto · Petri Myllymäki
Helsinki Institute for Information Technology HIIT, Department of Computer Science, University of Helsinki, Finland

1 Introduction

Since the formalization and popularization of Bayesian networks [53] for modeling and reasoning with multiple variables, much research has been devoted to learning them from statistical data [28]. One of the main challenges has been to learn the model structure, represented by a directed acyclic graph (DAG) on the variables. Cast as a problem of finding a DAG that is a global optimum of a score function for given data, the *Bayesian network structure learning* problem (BNSL) is notoriously NP-hard; the hardness is chiefly due to the acyclicity constraint imposed on the DAG to be learned [14]. To cope with the computational hardness, early work on structure learning resorted to local search algorithms. While local search algorithms oftentimes perform well, they are unfortunately unable to guarantee global optimality of produced networks. This uncertainty about the quality hampers the use of the network [48] in probabilistic inference and causal discovery.

The last decade has raised hopes of solving larger problem instances to optimality. The first algorithms guaranteed to find the optimum adopted a dynamic programming approach to avoid exhaustive search in the space of DAGs [51,36,61, 60]. Later algorithms have expedited the dynamic programming approaches using the A* search algorithm with various admissible heuristics [70], or have employed quite different approaches, such as branch and bound in the space of (cyclic) directed graphs [11], integer linear programming (ILP) [34,16,17], and constraint programming (CP) [5]. In this work, we focus on such *complete solvers* for BNSL, which we call simply *solvers*. Our interest is in unsupervised learning of a joint structure over the variables, only noting in passing that alternative methods have been developed for supervised learning of the relationship between a designated response variable and the other, predictor variables (see, e.g., a recent survey [7] and references therein).

Due to the intrinsic differences between the algorithmic approaches underlying BNSL solvers, it is not surprising that their relative efficiency varies greatly on a per-instance basis. To exemplify this, a comparison of the running times of three current state-of-the-art solver, based on A*, ILP, and CP, is illustrated in Figure 1 using typical benchmark datasets. Evidently, no single one of these three solvers dominates the two others.

Figure 1 suggests that, to improve over the existing solvers, an alternative to developing yet another solver is to design *hybrid* algorithms, or *algorithm portfolios*, which would ideally combine the best-case performance of the different solvers. Indeed, in this work we do not focus on developing or improving an individual algorithmic approach. Instead, we aim to characterize how the performance of different algorithmic approaches depends on the problem instance, which is the key to the design of efficient algorithm portfolios and hybrids. The underlying motivation for developing such techniques is the aim of improving the efficiency of state of the art in complete solvers in solving hard BNSL instances.

In this quest, it is vital to discover a collection of *features* that are efficient to compute and yet informative about the hardness of an instance for a solver. Prior work has identified two simple features, namely the number of variables and the number of so-called *candidate parent sets*, denoted by n and m, respectively. To explain the observed orthogonal performance characteristics shown in Figure 1, it has been suggested, roughly, that typical instances can be solved to optimality by A*, if n is at most 40 (no matter how large m), and by ILP if m is moderate, say, at

most some tens of thousands (no matter how large n) [17,70]; for the more recent CP approach, we are not aware of any comparable description. Beyond this rough characterization, the practical time complexity of the best-performing solvers is currently poorly understood. This stems from the sophisticated search heuristics employed by the solvers, which tend to be sensitive to small variations in the instances, thus resulting in somewhat chaotic-looking behavior of running times. Furthermore, the gap between the analytic worst-case and best-case running time bounds, in terms of n and m, is huge, and typical instances fall somewhere in between the two extremes.

The starting point of our work is the following basic open question:

Q1 For *determining the fastest* of the available solvers on a given instance, do the simple features, the number of variables and the number of candidate parent sets, suffice?

We answer this question in the affirmative. Our result is empirical in that it relies on training and testing a statistical model with a large set of problem instances collected from various sources. We show that a simple set of features gives yields a model which accurately predicts the fastest solver for a given instance based on the parameters n and m only. Furthermore, we show how this yields an algorithm portfolio that almost always runs as fast as the fastest solver, thus significantly outperforming any fixed solver on a large collection of instances.

However, a closer inspection of the model reveals that the running times it predicts often differ from the actual running times by one to two orders of magnitude. The large deviations suggest that, if the interest is in accurate estimation of the running times, then the simple feature set should probably be extended by additional features:

Q2 For *predicting the running time of a solver* on a given instance, can the accuracy be significantly improved by including additional efficiently computable features?

Also to this question our answer is affirmative. We introduce and study several additional features that potentially capture the hardness of the problem more accu-

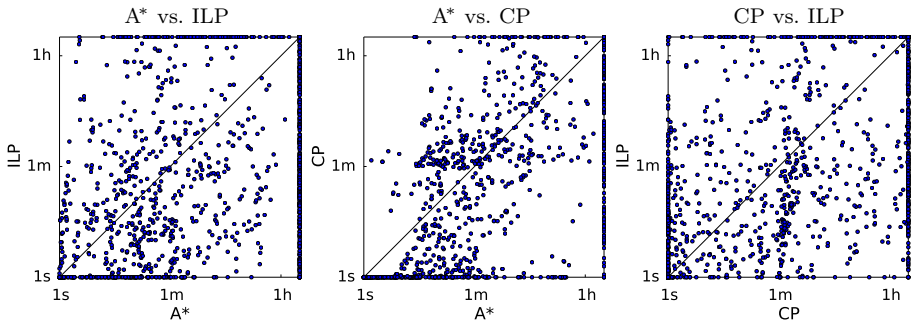


Fig. 1 Comparison of three state-of-the-art algorithms for finding an optimal Bayesian network. Running times below 1 or above 7200 seconds are rounded to 1 and 7200, respectively. The specific solver parameterizations are ILP: GOBNILP INSERT VERSION HERE!, A*: A*INSERT VERSION HERE!CP: CPBayes; see Section 5 for descriptions of the solvers and the datasets.

rately for a given solver. We focus on what are currently the three top-performing solver families based on A* [70], ILP [17], and CP [5], which clearly dominate earlier approaches based on dynamic programming and branch-and-bound [49]. Specifically, we show that models with a wider variety of features yield at times significant improvements in prediction accuracy.

Besides the aforementioned contributions, the empirical work associated with this paper also provides the most elaborate evaluation of state-of-the-art solvers to date, significant in its own right.

need to revise this paragraph after everything else! The present work extends and revises substantially our preliminary study reported at the AAAI-14 conference [49]. Here we have thoroughly revised the methodology and analysis presented throughout the paper. In particular, we have completely replaced the previous machine learning framework with the state-of-the-art auto-sklearn system [20]; it optimizes the choice of preprocessing, model class and model class parameters. Thus, we avoid difficult, ad hoc choices for these important characteristics. We have also expanded the portfolio itself to include the very recent CP-based solver [5]. At the same time, we have updated the runtime results to the most recent versions of the A*-based and ILP-based solvers. Furthermore, we provide a more fine-grained analysis by categorizing datasets based on their origin into benchmark and synthetic data sets. Our results show that the origin of the dataset significantly affects the relative solver performances. To this end, we have also increased the number of synthetic data sets considerably, from a few dozens to several hundred. Finally, we provide a more extensive discussion of the characteristics of the learned models, such as preprocessing strategies.

1.1 Related Work

Due to the wide range of potential applications, the general research area of algorithm selection, with tight connections to machine learning and algorithm portfolio design, is very diverse. Instead of aiming at a full review of the relevant literature, here we aim at a brief overview of the research area by providing references to some of the key early works on the topic and some of the more recent works most closely related to ours. For an expanded discussion of the literature on algorithm selection and runtime prediction, we refer the reader to two recent surveys on the topic with further pointers to related work [33, 38].

Research on algorithm selection for various types of important computational problems has its roots in [56], where the algorithm selection problem was already introduced, and feature-based modeling was proposed to facilitate the selection of the best-performing algorithm for a given problem instance, considering various example problems. Later works, including [12, 21, 46], demonstrated the efficacy of applying machine learning techniques to learn models from empirical performance data. A Bayesian approach was proposed e.g. in [30].

More recently, empirical hardness models [44, 45] have been applied in the construction of solver portfolios [26] for various NP-hard search problems [39], including Boolean satisfiability (SAT) (e.g. in [68]), constraint programming (e.g. in [24, 31]), quantified Boolean formula satisfiability (e.g. in [55]), answer set programming (e.g. in [29]), as well as for the traveling salesperson problem (e.g. in

[40]). To the best of our knowledge, for the important problem of Bayesian network structure learning, the present work is the first to adopt the approach.

In terms of terminology, we investigate algorithm selection in the context of learning Bayesian networks, which is an *unsupervised* learning task. Nevertheless, this work is well-situated in the context of meta-learning [25], which most often consider supervised settings. The BNSL features we propose in Section 3.1 are exactly a set of *meta-features* for this particular domain. The regression models we learn (Section 3.2) capture meta-knowledge about the state-of-the-art BNSL solvers.

Previous work [41,43] has suggested that in many cases, a small set of features can lead to accurate predictions; indeed, in Section 5.2 we show that a very small number of features leads to near-optimal algorithm selection performance. Furthermore, while that work relied on qualitative visual analysis, in Section 6.4 we quantify the utility of each feature using the Gini importance [9].

Recently, a simple "Best in Sample" approach [57] was shown to be very effective for algorithm (classifier) selection in the supervised setting. Briefly, this approach trains each classifier in the portfolio using a very small subset of the data; it then selects the classifier to use based on performance on the subset. "Probing" features—a central form of features in e.g. SAT portfolios [68]—we apply in the context of BNSL (see Section 3.1) are similar in spirit to this approach, though adapted to the unsupervised learning setting. In terms of evaluation, our virtual best solver comparisons in Section 5 are quite similar to Loss Curves [42], which have previously been used in the context of meta-learning.

1.2 Organization

The remainder of this paper is organized as follows. We begin in Section 2 by describing the problem of structure learning in Bayesian networks and by giving an overview of the algorithmic techniques underlying the state-of-the-art solvers. Section 3 presents the building blocks of our empirical hardness model: we introduce several BNSL features; we choose an appropriate statistical learning framework; and we describe the methods we use for training and evaluating the models. In Section 4, we present the experimental setting, namely technical details of the investigated solvers and characteristics of the collected problem instances. Results on learning solver portfolios and on predicting running times of individual solvers are reported in Sections 5 and 6, respectively. Finally, we discuss some questions that are left open and directions for future research in Section 7.

2 Learning Bayesian Networks

A Bayesian network (G, P) consists of a directed acyclic graph (DAG) G on a set of random variables $X_1, \ldots, X_n$ and a joint distribution P of the variables such that P factorizes into a product of the conditional distributions $P(X_i|G_i)$. Here G_i denotes the set of parents of X_i in G; we call a variable X_j a *parent* of X_i, and X_i a *child* of X_j, if G contains an arc from X_j to X_i.

2.1 The Structure Learning Problem

In its simplest form, structure learning in Bayesian networks concerns finding a DAG that best fits some observed data on the variables.[1] Throughout this work, we only deal with this optimization formulation, here only mentioning that there are also other popular formulations based on frequentist (multiple) hypothesis testing [63,13] and Bayesian model averaging [47,23,36].

The goodness of fit is typically measured by a real-valued scoring function s, which associates a DAG G with a real-valued score $s(G)$.[2] Frequently used scoring functions—based either on (penalized) maximum likelihood, minimum description length, or Bayesian principles (e.g., BDeu and other forms of marginal likelihood)—decompose into a sum of local scores $s_i(G_i)$ for each variable X_i and its set of parents G_i [28]. In principle, for each i the local scores are defined for all the 2^{n-1} possible parent sets. However, in practice this number is greatly reduced by enforcing a small upper bound for the size of the parent sets G_i or by pruning, as preprocessing, parent sets that provably cannot belong to an optimal DAG [65, 11]. Applying one or both of these reductions results in a collection of *candidate parent sets*, which we will denote by $\mathcal{G}_i$.

This motivates the following formulation of the *Bayesian network structure learning* problem (BNSL).

INPUT: Local scores $s_i(G_i)$ for a collection of candidate parent sets $G_i \in \mathcal{G}_i$ for $i = 1, \ldots, n$.
TASK: Find a DAG G that maximizes the score $s(G) = \sum_i s_i(G_i)$.

Along with the number of variables n, another key parameter describing the input size is the total number of candidate parent sets $m = \sum_i |\mathcal{G}_i|$. See Figure 2 for an example instance of the BNSL problem.

2.2 Overview of Complete Solvers for BNSL

We call an algorithm that solves the BNSL problem to a guaranteed global optimum a *complete solver* for BNSL, or simply a *solver*. In the next paragraphs we review some state-of-the-art solvers that fit the scope of our study. We omit algorithms that assume significant additional constraints given as input [54] or massive parallel processing [64,52].

Several works [51,36,60] have proposed dynamic programming algorithms to solve BNSL. The solvers are based on the early observation [10,15] that for any fixed ordering of the n variables, the decomposability of the score enables efficient optimization over all DAGs compatible with the ordering. The algorithms proceed by adding one variable at a time, only tabulating partial solutions for the explored *subsets* of the variables. Thus the running time scales roughly as 2^n.

[1] Strictly speaking, the data are assumed to consist of some number N of independent and identically distributed tuples $(X_1^t, \ldots, X_n^t)$, $t = 1, \ldots, N$, the dimension of the data being $N \times n$.

[2] The score does not depend on the parameters of the unspecified distribution P, which are treated as nuisance parameters and absorbed by the scoring function (e.g., estimated or integrated away).

Variable X_i	**Candidate Parents** $\mathcal{G}_i$	**Local Score** $s_i(G_i)$
X_1	$\emptyset$	**2.0**
X_2	$\emptyset$	**1.0**
X_3	$\emptyset$	0.2
X_3	$\{X_1\}$	**1.0**
X_4	$\emptyset$	**0.1**
X_4	$\{X_6\}$	0.8
X_5	$\emptyset$	0.1
X_5	$\{X_1\}$	0.7
X_5	$\{X_1, X_2\}$	**2.0**
X_6	$\emptyset$	0.2
X_6	$\{X_3\}$	0.8
X_6	$\{X_3, X_4\}$	**2.0**
X_7	$\emptyset$	0.1
X_7	$\{X_5\}$	0.5
X_7	$\{X_4, X_5\}$	**1.0**

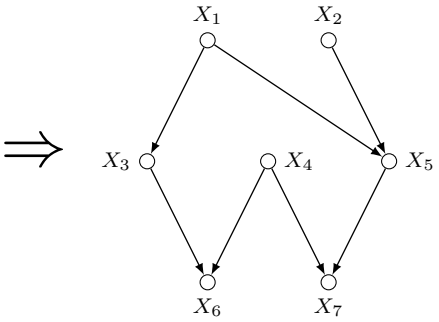


Fig. 2 An optimal DAG (on the right) for a given scoring function s (on the left). There are $n = 7$ variables and $m = 15$ candidate parent sets in total. The optimal score, 8.1, is the sum of the local scores shown in bold face. Observe that choosing $G_4 = \{X_6\}$ would have increased the score but violated the acyclicity constraint.

Yuan and Malone [70] formulated BNSL as a state space search through the dynamic programming lattice and applied the A* search algorithm. Unlike the other more sophisticated solvers, A* maintains the worst-case time bound of dynamic programming. To this end, they developed several admissible heuristics which relax the acyclicity constraint; these allow the algorithm to prune suboptimal paths during search, thus typically avoiding visiting all the variable subsets.

The branch-and-bound style algorithm by de Campos and Ji [11] searches in a relaxed space of directed graphs that may contain cycles. It begins by allowing all variables to choose their optimal parents, which typically results in some number of cycles. Then, any found cyclic solutions are iteratively ruled out: it finds a cycle and breaks it by removing one arc in it, branching over the possible choices of the arc. It examines graphs in a best-first order, so the first acyclic graph it finds is an optimal DAG. In this way, the algorithm ignores many cyclic graphs.

Integer linear programming (ILP) algorithms by Jaakkola et al. [34] and by Bartlett and Cussens [16,17] search in a geometric space, in which DAGs appear as vertices of an embedded polytope, corresponding to integral solutions to a linear program (LP). A series of LP relaxations are solved, and the solution to each relaxation is checked for integrality; an integral solution corresponds to an optimal DAG. The search space is effectively pruned by employing domain-specific cutting planes.

A very recent development in solvers for BNSL is the constraint programming (CP) based approach by van Beek and Hoffmann [5], constituting a constraint-based depth-first branch-and-bound approach to BNSL. As a key ingredient, the approach uses an improved constraint model with problem-specific dominance, symmetry breaking, and acyclicity constraints and propagators, as well as cost-based pruning rules applied during search, together with domain-specific search heuristics.

3 Empirical Hardness Models

In this work, we focus on the *hardness* of a BNSL instance, relative to a particular solver. We define the hardness of instance I for solver S simply as the running time $T_S(I)$ of the solver S on the instance I.[3] Due to the sophisticated heuristics underlying the state-of-the-art BNSL solvers, evaluating the empirical hardness is presumably (under standard complexity-theoretic assumptions) computationally intractable; indeed, the fastest method we are aware of for evaluating $T_S(I)$ is actually running S on I.

Rather than exactly evaluating the function T_S, we take a machine learning approach to approximate it: from a large collection of example instances for which the actual running times are known (computed), we learn a *model* which is efficient to evaluate at any given instance. Underlying this approach is the hypothesis that an accurate *empirical hardness model* [44] can be built based on a set of efficiently computable *features* of BNSL instances; by a feature we refer to a mapping from the instances to the real numbers. This approach naturally gives rise to the following supervised machine learning problem, for a fixed solver S.

INPUT:	A training set of BNSL instances (represented as collections of feature values) and the respective running times of the solver S.
TASK:	Learn a function $\hat{T}_S$ so as to minimize the average prediction error on an unseen set of BNSL instances.

We next introduce several categories of efficiently-computable features of BNSL instances. Most of these features have not previously been used for characterizing the hardness of BNSL. We then explain our training and testing strategies.

3.1 Features for BNSL

We use four different strategies to efficiently compute features to characterize BNSL instances: **Basic**, **Basic extended**, **Upper bounding**, and **Probing**. Table 1 lists the features in each category. Further, for the purpose of comparison, we define the category **All** as encompassing all features of all categories.

The **Basic** features are the number of variables n and the mean number of candidate parent sets per variable, m/n, which can be viewed as a natural measure of the "density" of an instance. The features in **Basic extended** are other simple features that summarize the size distribution of the collections $\mathcal{G}_i$ and the candidate parent sets G_i in each $\mathcal{G}_i$. During training, we take the logarithm of the features related to the number of candidate parent sets (Features 2 – 5).

In the **Upper bounding** category, the features are characteristics of a directed graph that is an optimal solution to a relaxation of the original BNSL problem. Notice here especially the features based on strongly connected components (SCCs), which can be seen as a proxy for cyclicity.[4] In the **Simple UB**

[3] While, in principle, the function T_S also depends on external factors such as the specific hardware on which the solver is run, we do not consider those factors in this work.

[4] Note that counting the number of cycles in a given graph is, in terms of computational complexity, presumably highly intractable, whereas SCC computation is achieved fast with well-known polynomial-time algorithms.

subcategory, a graph is obtained by letting each variable select its best parent set according to the scores. The resulting graph may contain cycles, and the associated score is a guaranteed upper bound on the score of an optimal DAG. Many of the reviewed state-of-the-art solvers either implicitly or explicitly use this upper bounding technique; however, they do not use this information to estimate the difficulty of a given instance. The features summarize structural properties of the graph: in- and out-degree distribution over the variables, and the number and size of non-trivial strongly connected components. In the **Pattern database UB** subcategory, the features are the same but the graph is obtained by solving a more sophisticated relaxation of the BNSL problem using the *pattern databases* technique [69]. Briefly, this strategy optimally breaks cycles among some subsets

Table 1 BNSL features

Basic
1. **Number of variables**
2. **Mean number of CPSs** (candidate parent sets)

Basic extended
3–5. **Number of CPSs** max, sum, sd (standard deviation)
6–8. **CPS cardinalities** max, mean, sd

Upper bounding

Simple UB
9–11. **Node in-degree** max, mean, sd
12–14. **Node out-degree** max, mean, sd
15–17. **Node degree** max, mean, sd
18. **Number of root nodes** (no parents)
19. **Number of leaf nodes** (no children)
20. **Number of non-trivial SCCs** (strongly connected components)
21–23. **Size of non-trivial SCCs** max, mean, sd

Pattern database UB
24–38. The same features as for *Simple UB* but calculated on the graph derived from the pattern databases

Probing

Greedy probing
39–41. **Node in-degree** max, mean, sd
42–44. **Node out-degree** max, mean, sd
45–47. **Node degree** max, mean, sd
48. **Number of root nodes**
49. **Number of leaf nodes**
50. **Error bound**, derived from the score of the graph and the pattern database upper bound

A probing*
51–62. The same features as for *Greedy probing* but calculated on the graph learned with A* probing

ILP probing
63–74. The same features as for *Greedy probing* but calculated on the graph learned with ILP probing

CP probing
75–86. The same features as for *Greedy probing* but calculated on the graph learned with CP probing

of variables but allows cycles among larger groups; it is a strictly tighter relaxation than the **Simple UB**. Both A* and CP explicitly make use of the pattern database relaxation.

Probing refers to running a solver for a fixed number of seconds and collecting statistics about its behavior during the run. Probing has previously been shown to be a central form of features e.g. in the context of Boolean satisfiability within the SATzilla portfolio approach [68]. For a survey on uses of probing features in other domains, see [33]. Here in the context of BNSL we consider four probing strategies: greedy hill climbing with a TABU list and random restarts, an anytime variant of A* [50], and the default versions of ILP [17] and CP [5]. All of these algorithms have anytime characteristics, so they can be stopped at any time and output the best DAG found so far. Furthermore, the A*, ILP and CP implementations give guaranteed error bounds on the quality of the found DAGs in terms of the BNSL objective function; an error bound can also be calculated for the DAG found using greedy hill climbing by using the upper bounding techniques discussed above. Probing is implemented in practice by running each algorithm for 5 seconds and then collecting several features, including in- and out-degree statistics and error bound. We refer to these feature subcategories of **Probing** as **Greedy probing**, **A* probing**, **ILP probing**, and **CP probing**, respectively.

3.2 Model Training and Evaluation

We use the auto-sklearn system [20] to learn an empirical hardness model $\hat{T}_S$ for each solver S; we refer to the highly sophisticated machine learning algorithm implemented in auto-sklearn, simply as a *learner*. Briefly, auto-sklearn uses a Bayesian optimization strategy for learning good model classes and hyperparameters for those model classes; additionally, preprocessing strategies, such as polynomial expansion or feature selection, and associated hyperparameters are included in this optimization. Importantly, this approach avoids the difficult step of manually choosing hyperparameters in an ad hoc fashion. In total, the learner selects among eleven preprocessing strategies, including higher dimensional projection techniques like polynomial expansion and feature selection strategies based on, for example, mutual information; twelve model classes for regression are used in the optimization, including random forests, several support vector machine-based strategies and various generalized linear models with different regularization strategies. We refer the reader to the original publication [20] for more details.

As described in detail in Section 4.2, this study includes three types of BNSL instances: Real, Sampled and Synthetic. For training, we mix instances of all types of datasets. As a first step in training, we normalize each feature so that it has zero mean and unit variance; the same mean and variance are later used to scale the test data. We then use auto-sklearn to learn two different, independent regression models for each solver. First, we optimize prediction accuracy by learning an ensemble of 50 regressors with optimized hyperparameters; each regressor also has its own preprocessing technique. We largely treat the ensemble as a blackbox model for regression. Second, we use auto-sklearn to find a single random forest (and associated preprocessor) with optimized hyperparameters; in this setting, the goal is to find an interpretable model to allow qualitative analysis.

The portfolios and prediction accuracy are evaluated using standard 10-fold cross-validation. In other words, the data is partitioned into 10 non-overlapping subsets. For each fold, nine of the subsets are used to train the model. Internally, auto-sklearn uses one-third of the training data as a validation set. We give 5 hours for training time for each fold. The remaining subset is used for testing, which only takes a few seconds; each subset is used as the testing set once. The instances are partitioned the same way for all solvers. We predict the runtime of each testing instance using the appropriate model for each solver. We then either select the solver with the lowest predicted runtime (and examine its actual runtime against the runtime of the fastest solver) or compare the predicted runtimes to the actual runtimes. The overall performance is the union of performance on each test subset.

4 Experimental Setup

We continue with a detailed description of our experimental setup, including descriptions of the solver parameterizations used, the data sets used in the experiments, as well as the computing infrastructure used.

4.1 Solvers

Our focus is on *complete* BNSL solvers that are capable of producing guaranteed globally optimal solutions. Specifically, we evaluate three complete approaches: Integer-Linear Programming (ILP), A*-based state-space search (A*), and a constraint programming based approach (CP). Importantly, these approaches constitute the current state-of-the-art solvers for BNSL.[5]

We consider the following solvers and their parameterizations.

ILP We use the GOBNILP solver [17] as a state-of-the-art representative of the ILP-based approaches to BNSL. GOBNILP uses the SCIP framework [1] and an external linear program solver; we chose the open source SoPlex solver [67] bundled with the SCIP Optimization Suite. We consider the most recent version, GOBNILP 1.6.2, which uses SCIP 3.2.0 with Soplex 2.2.0, as well as GOBNILP 1.4.1 (SCIP 3.0.1, SoPlex 1.7.1). For both versions we consider two parameterizations: the default configuration, which searches for BNSL-specific cutting planes using graph-based cycle finding algorithms, and a second configuration, "-nc" ("no cycle-finding"), which only uses nested integer programs. We call these parameterizations ilp-141, ilp-141-nc, ilp-162, and ilp-162-nc, respectively, for short.

A* We use the URLearning solver [70] as a state-of-the-art representative approach to BNSL based on the well-known A* search method. We consider three variants: A*-ed3, which uses dynamic pattern databases, A*-ec, which uses a combination of dynamic and static pattern databases, and A*-comp which uses a strongly connected component-based decomposition [18].

[5] In a preliminary version of this work [49], we also considered an earlier proposed branch-and-bound approach [11], which we found to be always dominated by ILP; therefore, we dropped it from consideration. Furthermore, the earlier proposed dynamic programming approach [36] is clearly dominated by A*. We have also discarded some parameterizations of both ILP- and A*-based solvers that were found to be uncompetitive.

CP We use the CPBayes solver [5] as the most recent state-of-the-art representative approach to BNSL based on branch-and-bound style constraint programming search with problem-specific filtering (search-space pruning) techniques. This solver does not expose any parameters to control its behavior. As such we apply the solver in our experiments in its default configuration, `cpbayes`.

The non-default parameterizations of the solvers were suggested to us by the solver developers. While we use both an "up-to-date" version (1.6.2) and an older version (1.4.1) of GOBNILP, it is important to note that, generally, the choice of parameters and the solver version can at times have a noticeable effect on the per-instance running times of the resulting solver—so much so that one could consider the solvers different.[6]

4.2 Training Data

To train our models we first obtained a collection of datasets from various sources. For each dataset we then evaluated one or more scoring functions to produce a collection of BNSL instances. We used datasets from the following three categories.[7]

REAL Real-world datasets obtained from machine learning repositories: the UCI repository [2], the MLData repository (`http://mldata.org/`), and the Weka distribution [27]. We searched primarily for datasets of fully or mostly categorical data and a reasonable number of variables (16–64) to produce instances that are feasible but non-trivial to solve. Every dataset found and matching these criteria was included. While some of the datasets have originally been designed for supervised learning, they have been regularly included also in studies of unsupervised learning. These datasets are summarized in more detail in Table 7 of Appendix A.

SAMPLED Datasets sampled from benchmark Bayesian networks, obtained from `http://www.cs.york.ac.uk/aig/sw/gobnilp/`. These datasets are widely used for evaluating the performance of individual solvers, for example, recently in the context of optimal BNSL in [4–6, 17–19, 49, 48, 58]. These datasets are summarized in Table 8 of Appendix A.

SYNTHETIC Datasets sampled from synthetic Bayesian networks. We generated random networks of varying number of binary variables (20–60) and maximum in-degree (2–8). For each network one dataset was produced by sampling a random number (100–10,000) of records.

We preprocessed each dataset by removing unique identifiers (to avoid overfitting) and trivial variables that only take on one value. Continuous variables as

[6] For corroborating evidence on this, see e.g. empirical data provided in [17] for different parameterizations and versions of GOBNILP.

[7] The main motivations for including both more real and, on the other hand, synthetic datasets in the study are two-fold: (i) We aimed at a notably heterogeneous set of benchmarks for the study, yielding insights into the prediction task on a wide range of datasets with different properties; and (ii) the three-way categorization has analogies in the benchmark categorization used in the SAT domain [35].

Table 2 Number of source datasets, instances generated from the source datasets, and instances used in training and testing the models.

Category	Datasets	All Instances	Training & Testing
Real	39	637	486
Sampled	19	317	283
Synthetic	477	477	410

well as other variables with very large domains were either removed or discretized using a normalized maximum likelihood approach [37] when possible. The maximum number of records per dataset was limited to $60,000$ to make the evaluation of scoring functions feasible.

We considered five different scoring functions[8]: BDeu with the Equivalent Sample Size parameter selected from $\{0.1, 1, 10, 100\}$ and the BIC scores. For each dataset of the first two classes we produced multiple instances by considering all scoring functions and varying upper bounds on the size of each candidate parent set, ranging from 2 to 6, as well as the unbounded case. For each synthetic dataset we produced one instance, choosing both the scoring function and the parent limit at random. For larger datasets evaluating the scores was feasible only up to lower parent limits. The total number of datasets and BNSL instances produced is summarized in Table 2.

For running all solvers on these instances we used a cluster of Dell PowerEdge M610 computing nodes equipped with two 2.53-GHz Intel Xeon E5540 CPUs and 32-GB RAM. For each individual run we used one CPU core, with a timeout of two hours and a 30-GB memory limit. We treat the runtime of any instance as two hours if a solver exceeds either the time or memory limit.

For training the models we used a subset of all instances, obtained by removing very easy instances, solved within five seconds by all solvers, as well as instances on which all solvers failed.[9] We call these the *training* instances (see Table 2) and focus on them in the following sections.

4.3 Feature computation

In order to train the models we computed the features detailed in Section 3.1 for all training instances. Table 3 summarizes the time spent to compute these features separately for each feature category. We observe that the computation takes around 16 seconds per instance on average and about 26 seconds in the worst case. Further, most of the time is spent on probing, while features of all other categories are computed in less than one second. Probing occasionally requires more than the limit of 5 seconds to finish a preprocessing step, in which case it is given 10 seconds in total.

[8] In our experiments, the results were not very sensitive to the scoring function, except through its effect on the number of CPSs and other features, so our results can generalize to other decomposable scores as well.

[9] This is in line with related work on portfolio construction in other domains such as SAT [33] as well as the SAT Competitions where a similar criterion is used to filter out "too easy" instances from the competition benchmark sets [3]. Solver selection for very easy instances is essentially trivial, as any choice of a solver is essentially a good one.

Table 3 The running time of feature computation for each feature category in seconds, shown as the average, median, minimum, and maximum running time over all training instances.

Feature set	Average	Median	Min	Max
Basic	0.00	0	0	0
Basic extended	0.00	0	0	0
Lower bounding	0.00	0	0	0
Greedy probing	2.53	2	0	6
A* probing	4.61	5	0	7
ILP probing	3.94	5	0	10
CP probing	4.49	6	0	10
All	15.57	18	0	26

We conclude that the overhead from computing the features is negligible from a portfolio perspective, as our main interest is in choosing the fastest solver for harder instances that take several minutes or even hours to solve. The easiest instances by contrast are often solved already in the probing phase.[10]

4.4 Availability of Experiment Data

To facilitate open access and further analysis of the data produced in the experiments of this work, we have made the full solver running time data, as well as the models learned for runtime prediction, available at

`http://bnportfolio.cs.helsinki.fi/`

Furthermore, the runtime and feature data available as a scenario in the ASlib Algorithm Selection Library [8] for further benchmarking purposes at

`http://INSERT-URL-HERE and also to the response / KUSTAA`

5 Portfolios for BNSL

This section focuses on the construction of practical BNSL solver portfolios in order to address question Q1. Optimal portfolio behavior is to always select the best-performing solver for a given instance. As the main result, we will show that, perhaps somewhat surprisingly, it is possible to construct a practical BNSL solver portfolio that vastly outperforms any single solver using only the **Basic** features.

5.1 Solver Performance

As the basis of this work, we ran all the solver parameterizations on all the BNSL instances, as described in Section 4. A comparison of solver performance is shown in Figure 3, in terms of the number of instances for which a particular solver was empirically faster than all other solvers on the considered benchmarks. Table 4 shows an alternative comparison in terms of the total number of instances that were successfully solved within the given computational resources as well as the

[10] We note here that the benchmark set used was not filtered based on probing results.

total CPU time required to either solve an instance or run out of time or memory. We observe that among the ILP parameterizations the two default configurations, ilp-141 and ilp-162, are empirically the best performing on the considered benchmarks, while in terms of total running time all four show fairly similar performance empirically. Among the A* variants, A*-comp does best on average, while A*-ec outperforms A*-ed3 on nearly all instances and also A*-comp in the REAL class. The numbers are given in comparison to the Virtual Best Solver (VBS), which is the theoretically optimal portfolio that always selects the best algorithm, constructed by selecting *a posteriori* the fastest solver for each input instance; essentially, an upper bound on the runtime of any portfolio approach using k solvers is k times the running time of the VBS.

We proceed to present two real portfolios, portfolio-basic and portfolio-all, which use all eight solver parameterizations. As some of these parameterizations are highly correlated, we will from hereon present detailed comparisons only for a smaller set of representative solvers. To choose these representatives, we consider the Shapley value [59], which was recently proposed as a principled measure of a solver's contribution to a portfolio [22]. In this framework, one considers constructing a portfolio by adding solvers incrementally and measuring the value of each solver as the increase in the portfolio's performance when the solver is added. As these values greatly depend on the order in which solvers are added, the Shapley value of a solver is defined as its average value over all possible solver permutations. Table 5 shows the Shapley values for all solver parameterizations, using the total number of instances solved as the measure of portfolio performance. In the light of these results, we will focus on ilp-162, cpbayes, and A*-comp, since these solvers have the highest Shapley values among each solver family on the considered benchmarks.

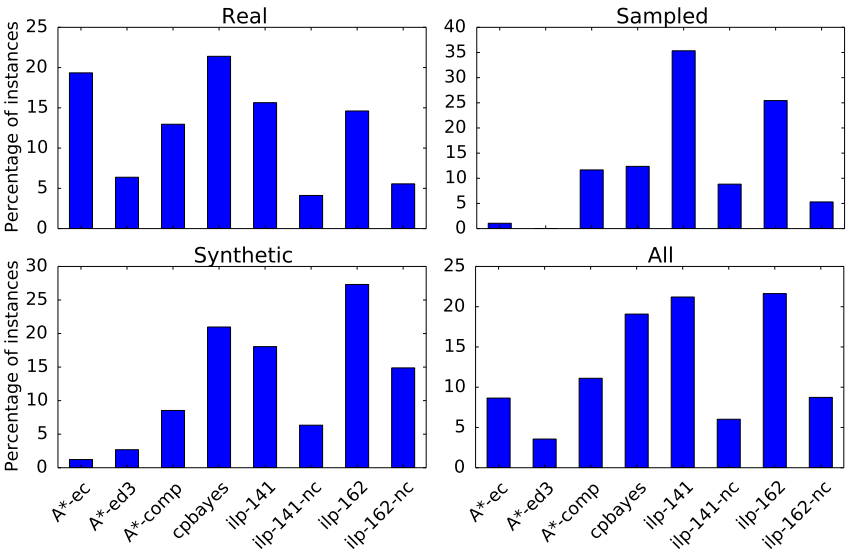


Fig. 3 The number of training instances for which a solver was fastest. Ties between solvers are broken at random.

Table 4 The performance of all solver parameterizations as well as the Virtual Best Solver (VBS) and two portfolios on all training instances, measured as the number of instances solved and the overall running time. Instances that were not successfully solved within the given resources count as 7200 seconds in the running times.

		Running time (s)		
Algorithm	**Instances solved**	**Cumulative**	**Average**	**Median**
VBS	1179	259,440	220	7.33
VBS without CP	1164	368,690	313	9.40
VBS without A*	1157	475,032	403	8.96
VBS without ILP	937	2,022,296	1,715	33.35
portfolio-all	1152	459,765	390	9.02
portfolio-basic	1134	569,351	483	11.95
ilp-141	1036	1,364,855	1,158	36.39
ilp-141-nc	1034	1,384,022	1,174	41.83
ilp-162	1029	1,453,932	1,233	29.56
ilp-162-nc	1026	1,494,879	1,268	32.18
cpbayes	896	2,423,547	2,056	85.83
A*-comp	768	3,152,809	2,674	185.79
A*-ec	519	4,866,797	4,128	7,200.00
A*-ed3	478	5,163,876	4,380	7,200.00

While the ILP approach appears to be the best-performing measured in the total running time and the number of instances solved on the set of benchmarks considered, the results suggest that the performance of ILP on a per-instance basis is quite orthogonal to that of both CP and A* (recall Figure 1). We will now show that a simple BNSL solver portfolio can closely capture the best-case performance of *all eight* of the considered solvers and variants in terms of empirical runtimes.

5.2 A Very Simple Solver Portfolio

We found that using only the **Basic** features (number of variables, n, and mean number of candidate parent sets, m/n) is enough to construct a highly efficient BNSL solver portfolio. We emphasize that, while on an intuitive level the importance of these two features may be to some extent unsurprising, such intuition does not directly translate into an actual predictor that would close-to-optimally predict the best-performing solver.

Table 4 shows the performance of our portfolios compared to each individual solver parameterization as well as the Virtual Best Solver. Here, portfolio-basic is our simple portfolio, which uses only the **Basic** features to choose a solver, constructed and evaluated as described in Section 3.2. For comparison, we present results also for portfolio-all, which uses **All** features, that is, all features of all categories. Figure 4 presents a more detailed view of portfolio performance, measured in the number of instances solved within a specific time.

We observe that, while there remains some room for improvement compared to the VBS, portfolio-basic performs nearly as well as portfolio-all and greatly outperforms every individual solver.

For more in-depth understanding, Figure 5 gives more insight into the effect of the **Basic** features on the solver runtimes. The correlation between ILP and the number of candidate parent sets is most apparent (coefficient of determination,

Table 5 The contribution of each solver to the VBS and the two portfolios measured as the Shapley value.

Algorithm	VBS	portfolio-all	portfolio-basic
ilp-162	184.53	182.60	177.62
ilp-141	184.12	181.31	180.69
ilp-141-nc	182.48	180.46	179.16
ilp-162-nc	181.50	177.44	177.72
cpbayes	160.42	152.73	147.55
A*-comp	136.24	129.73	125.55
A*-ec	78.28	77.22	75.85
A*-ed3	71.43	70.50	69.86

that is, explained variance, $R^2 > 0.7$ for all variants[11]), while CP ($R^2 \approx 0.38$) and A* ($R^2 > 0.47$ for all variants) exhibit moderate correlation with the number of variables. Figure 6 highlights the advantages of different solver families in the space spanned by these two features. The figures support the rough characterization (recall Section 1) of the computational limitations of state-of-the-art solvers. Specifically, we observe that ILP can fairly reliably solve instances up to around $1,000$ candidate parent sets per variable and typically fails on instances beyond

[11] R^2 ranges from 0 to 1, where 0 indicates that the feature is completely uninformative about runtime, and 1 indicates that all of the variance in runtime is explained by the respective feature.

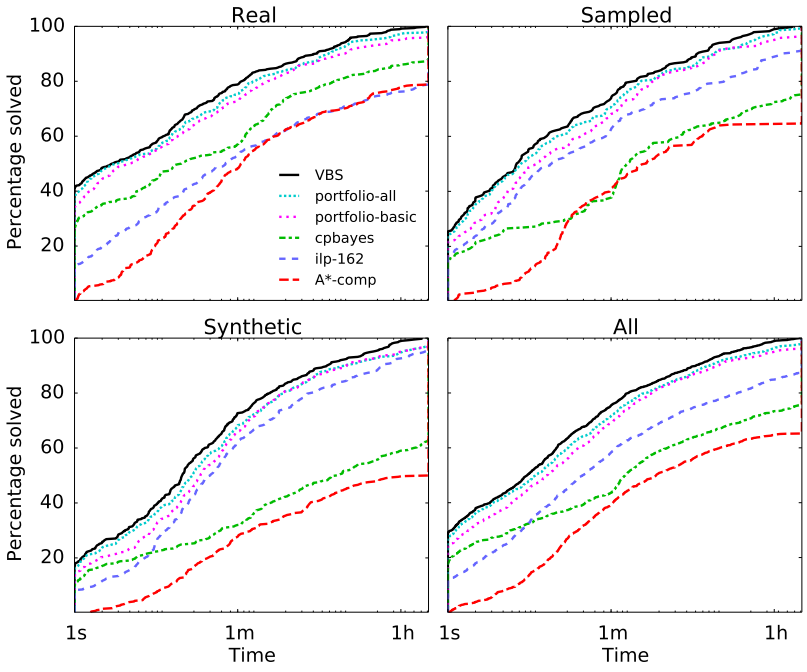


Fig. 4 Fraction of instances solved by the VBS, the portfolios, and individual solvers within a given amount of time.

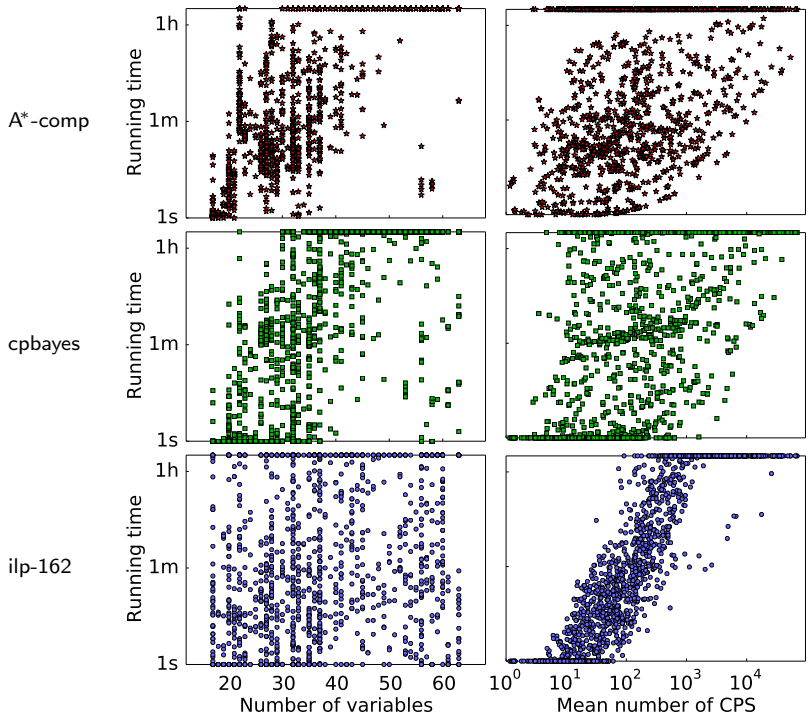


Fig. 5 Correlation between the **Basic** features and the runtimes of solvers.

this point. On the other hand, ILP appears not to depend heavily on the number of variables ($R^2 < 0.001$ for all variants), achieving a consistent performance throughout the spectrum considered in the benchmarks.

A*, by contrast, seems able to solve instances irrespective of the number of candidate parent sets ($R^2 < 0.05$ for all variants) but is in turn more heavily restricted by the number of variables; the benchmark instances are solved reliably up to 30 variables, many of them up to 40, and only very few past this point.

The CP approach appears to take a middle ground between the two extremes, solving many instances at the high end of either of the **Basic** features, albeit less consistently than A* and ILP. In terms of time required to solve an instance, CP appears to excel on instances where both the number of variables and candidate parent sets ($R^2 \approx 0.09$) are moderate, whereas A* and ILP typically do better than other solvers when the respective feature is large.

None of the state-of-the art solvers considered are able to solve the benchmark instances where both of the **Basic** features are large.

6 Predicting Runtimes

In this section we turn to the arguably harder problem of predicting per-instance runtimes of individual solvers. Apart from pure scientific interest, accurate runtime

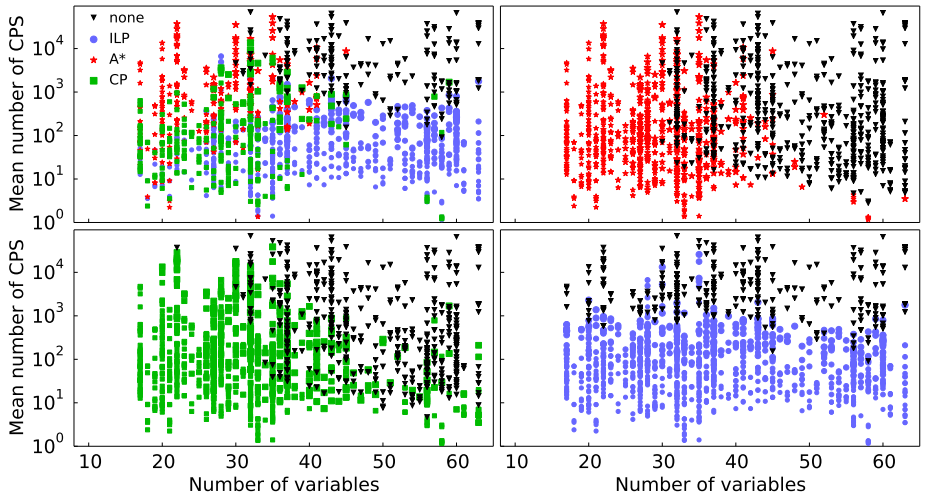


Fig. 6 Top left: All benchmark instances plotted in the space of the two **Basic** features, the number of variables and the mean number of candidate parent sets. Each instance is marked according to which solver was the fastest to solve it, specifically, whether the fastest solver was from the A*, ILP, or CP family, or whether none of the solvers could solve the instance. The other three plots present the same view separately for each solver family, highlighting their limitations as either or both features grow too large. Marker size scales as a logarithm of running time for instances that were successfully solved.

predictions on a per-instance basis are useful for job schedulers as computing clusters often require an estimated job time. In our case specifically, such predictions could also facilitate development of improved BNSL solvers. For example, a model could be exploited as a heuristic estimate for subproblem hardness during search within a parallel BNSL solver. As a further motivation, model-based algorithm configuration [32] crucially relies on runtime predictions in order to guide search for better configurations in the algorithm configuration space. In such contexts, note also that runtime is a primary resource to predict, as e.g. running out of other resources such as memory directly imply running out of time as well.

As shown in Section 5, the **Basic** features can effectively distinguish between solvers to use on a particular instance of BNSL. We will now address question Q2, that is, whether the accuracy of running time predictions can be improved with additional features (cf. Section 3.1).

6.1 Predictions with Added Features

Figure 7 depicts the actual runtimes of solvers compared to the runtimes predicted by the regression models based on ensembles learned with auto-sklearn. On the left we see this comparison for models trained using the **Basic** features only. Even though these predictions allow for good portfolio behavior, the considerable amount of prediction error makes them less useful for obtaining actual estimates of the runtime. The right side, on the other hand, shows the same comparison when using **All**, where the predictions are now more concentrated near the diagonal. In

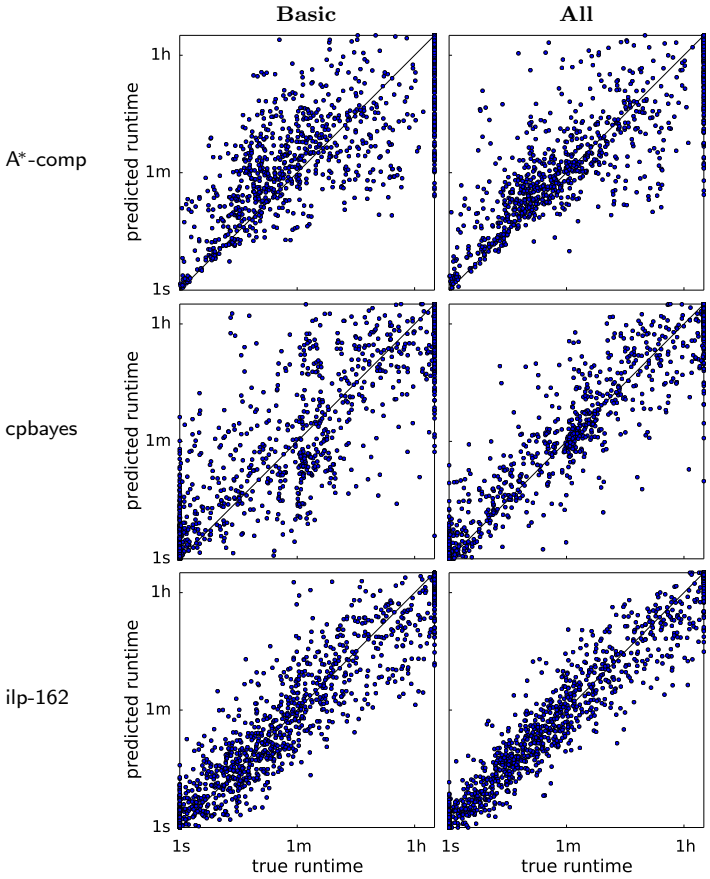


Fig. 7 The actual runtimes of solvers compared to the predicted runtimes when using the **Basic** (left) or **All** (right) features.

other words, the larger, more sophisticated feature set results in more accurate runtime predictions. Table 6 presents a numerical measure of the improvement in terms of change in the approximation factor, defined as $\rho = \max\{\frac{a}{p}, \frac{p}{a}\}$, where a and p are the actual and predicted runtimes, respectively. In particular, smaller approximation factors are better.

Table 6 The percentage of instances with an approximation factor within the given ranges of ρ, when predicting runtimes based on either **Basic** or **All** features. Higher percentages with lower approximation values indicate more accurate predictions.

	A*-comp		cpbayes		ilp-162	
Range of ρ	**Basic**	**All**	**Basic**	**All**	**Basic**	**All**
< 2	49%	62%	44%	68%	58%	72%
$[2, 5)$	21%	21%	28%	21%	30%	22%
$[5, 10)$	12%	7%	13%	6%	8%	4%
> 10	18%	10%	15%	4%	4%	1%

We also evaluated the impact of incrementally adding sets of features. Figures 8 and 9 show how the prediction error changes as we add **Basic** (features 1–2), **Basic extended** (1–23), **Upper bounding** (1–38), the relevant probing features for A* (1–38, 51–62), CP (1–38, 75–86), and ILP (1–38, 63–74), and finally **All** (1–86) for every solver. The results show that predictions using the **Basic** features are typically worse than those incorporating the other features, although this behavior is more pronounced for some solvers, feature sets and instance classes than others. The plots also suggest that some features help more than others for the different algorithms. For instance, **Upper bounding** features greatly improve the predictions of A* compared to the **Basic** and **Basic extended** features. By hindsight, this is relatively unsurprising since the efficacy of the upper bounding directly impacts the performance of A*, showing that the learner effectively exploits features we intuitively expect to characterize the empirical hardness. Probing offers a glimpse at the true runtime behavior of the algorithms, and the learner leverages this information to further improve prediction accuracy. For both A* and ILP probing with the respective solvers alone is informative, while the other probing strategies (**All** features) yield little improvement and even weaken some of the predictions. In contrast, surprisingly, for CP the predictions modestly benefit from probing with other solvers as well. Out of the three solver families CP predictions improve most from added features in general.

Finally, we evaluate the root mean squared error (RMSE) of the predictions for each solver as we incrementally add feature sets. Figure 10 echoes the results from Figures 8 and 9. We again see that **Upper bounding** improves predictions on all A* variants. The respective probing features greatly improve the prediction accuracy for A*-ec and A*-ed3; probing also improves the accuracy for the ILP family of solvers and cpbayes. Strikingly, the RMSE for A*-comp *increases* as the more sophisticated feature sets are added. As we show in Section 6.2, it appears the learner discards useful information from these features in preprocessing.

6.2 Preprocessing Characteristics

We now turn to more qualitative analysis based on a single random forest with optimized hyperparameters learned by auto-sklearn.

First, we examine preprocessor choices. As shown in Figure 11, the choice of preprocessor often reflects the amount of information inherently available in the feature sets. Furthermore, in the clustering, we see that the families of solvers tend to cluster together.

For the **Basic** feature set (dark tan), the learner almost always selects a preprocessor which increases the dimensionality, either the polynomial expansion or random forest embedding technique; we interpret this to mean that the features alone do not provide sufficient information for accurate prediction, so the learner attempts to increase the information with preprocessing. These are not always statistically significant, though, because the learner uses each of polynomial expansion and random forest embedding about half of the time. Likewise, many of the "mildly informative" feature sets, such as **Simple UB** (dark teal), almost exclusively result in polynomial expansion for preprocessing the input features. Interestingly, the **Basic extended** feature set (light tan) results in polynomial

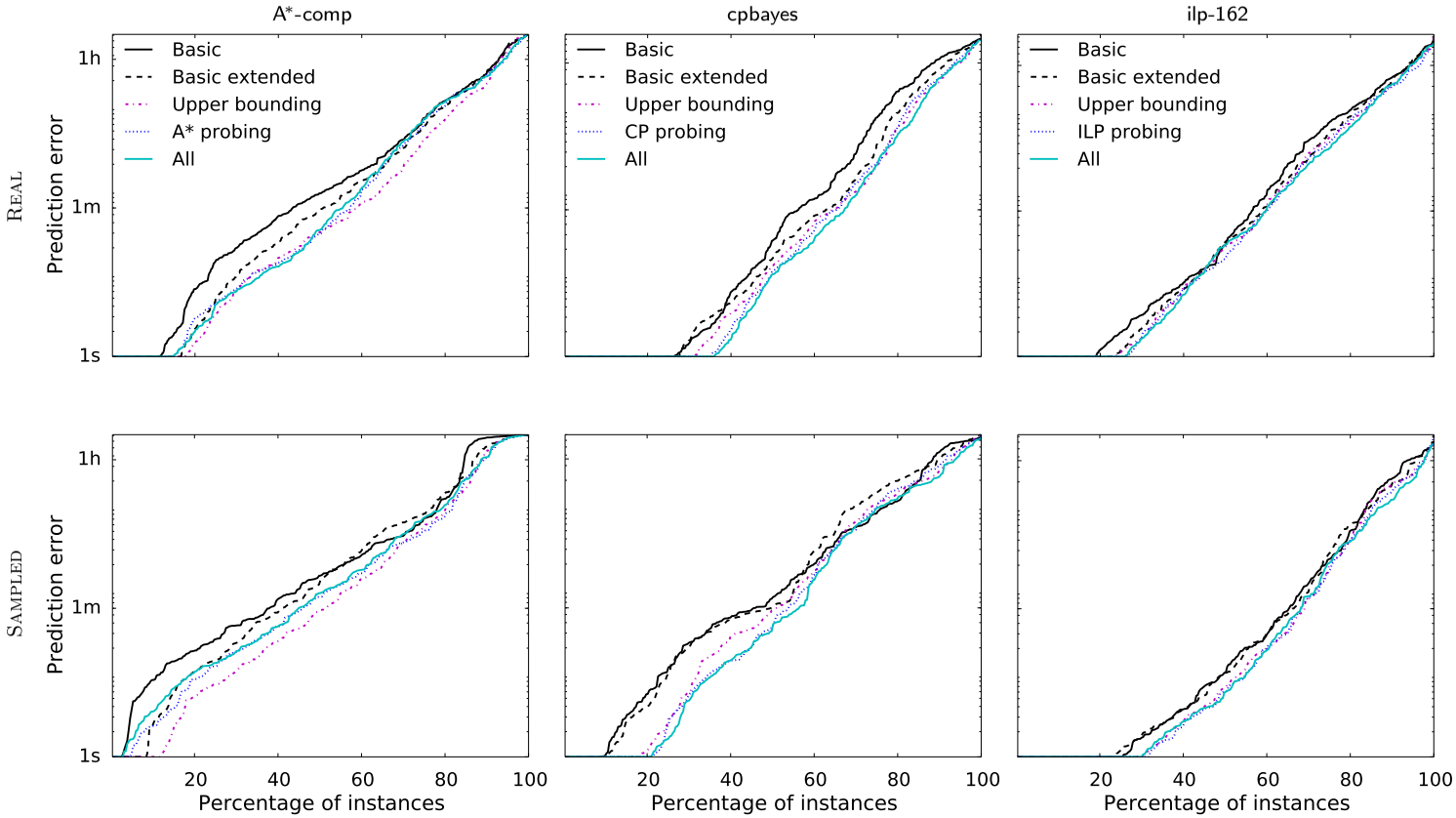


Fig. 8 The absolute prediction errors on REAL and SAMPLED instances using different sets of features, sorted in increasing order.

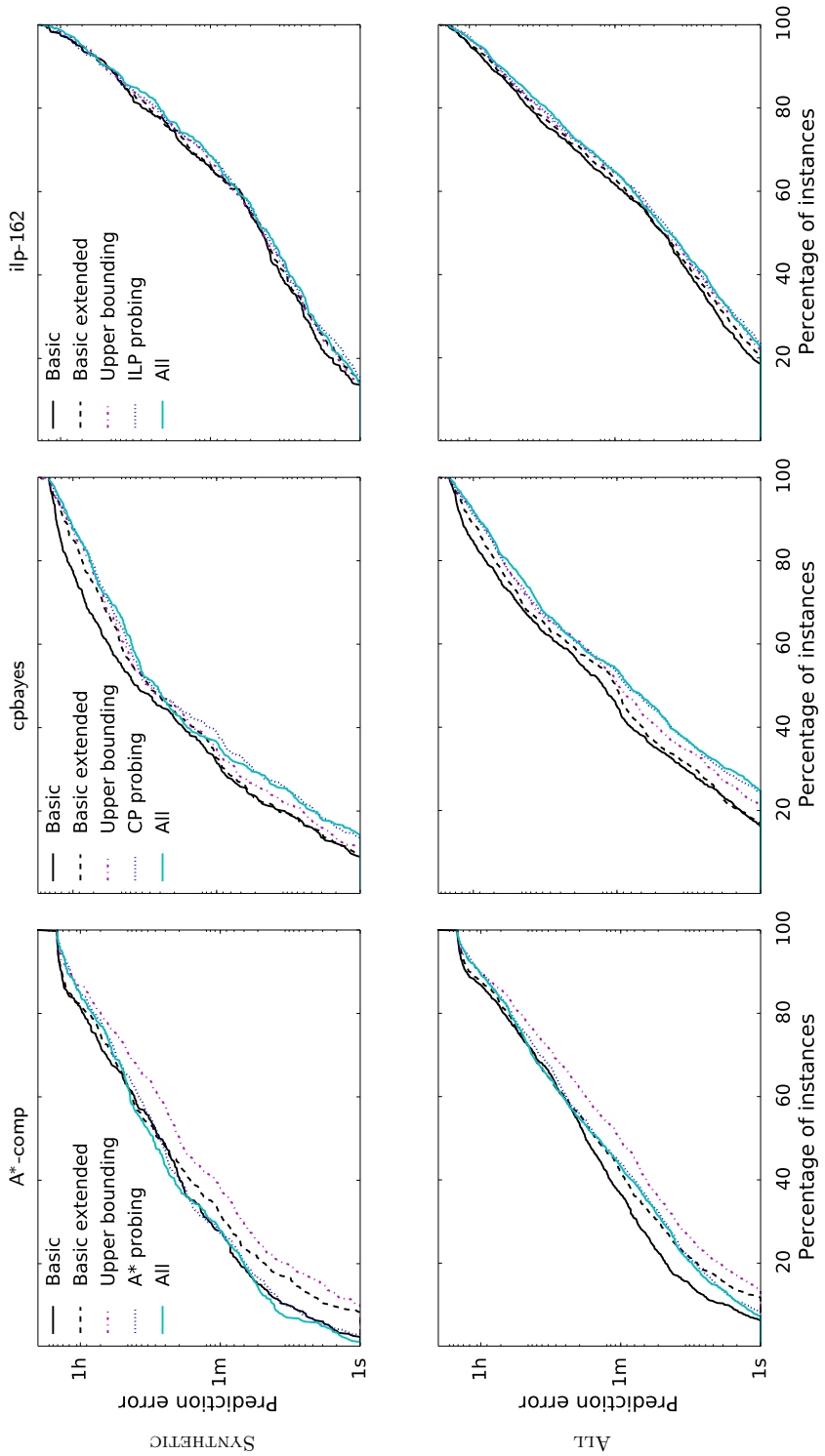


Fig. 9 The absolute prediction errors Synthetic and All instances using different sets of features, sorted in increasing order.

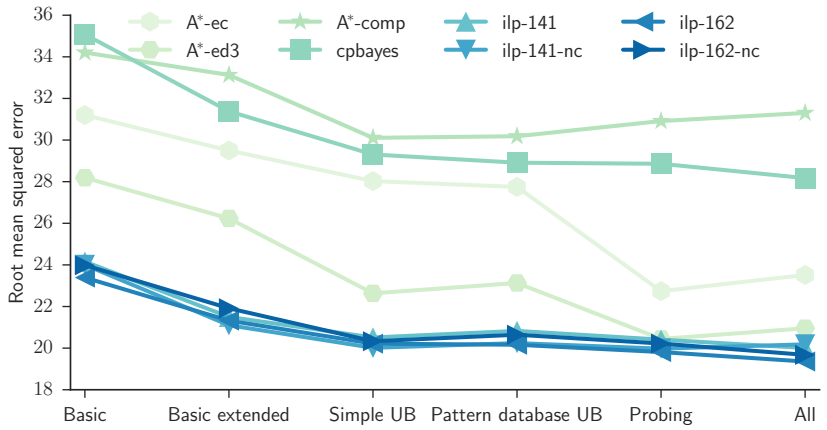


Fig. 10 The improvement of the root mean squared error of the runtime predictions as the more sophisticated features are used. "Probing" refers to the appropriate probing feature set for the respective solver, such as **A* probing** for the A*-ec solver.

expansion, a dimensionality expansion strategy, and feature agglomeration, a dimensionality reduction strategy, in roughly equal proportions for all solvers.

On the other hand, for the A* algorithms with the larger feature sets like **All** (light brown), the learner has "too much" information, so it uses feature aggregation, as well as model-based and percentile-based feature selection, to combine or remove uninformative features; these choices typically are statistically significant. For predicting most of the ILP runtimes using "informative" feature sets, such as **All** and **ILP Probing** (light teal), the learner does not typically use any preprocessing; again, almost all of these choices are statistically significant.

This analysis demonstrates that the choice of preprocessing strategy by auto-sklearn largely agrees with intuition. For small, relatively uninformative feature sets, feature expansion strategies like polynomial expansion are often used; when more informative features are available, the learner leaves them relatively unchanged. Finally, when "too much" information is present, the learner uses more sophisticated feature selection strategies to distinguish useful features from noise.

6.3 Model Complexity

We additionally analyzed the complexity of the learned random forests, in terms of the mean size of the regression trees composing them. As expected, Figure 12 (a) shows that the trees learned using the **Basic** features are the smallest. Other simpler feature sets, such as **Basic extended** and **Simple UB** also resulted in small trees for all solvers.

Somewhat surprisingly, though, the regression trees for the various ILP solvers are much larger than those for the cpbayes and A* family of solvers for the **A* probing**, **Pattern database UB**, **All** and **CP probing** feature sets. As shown in Figure 11, the learner often forewent preprocessing in these cases for ILP. On the other hand, it used sophisticated preprocessing, like the model-based approach, for A* and cpbayes a significant amount of the time. Thus, these results suggest

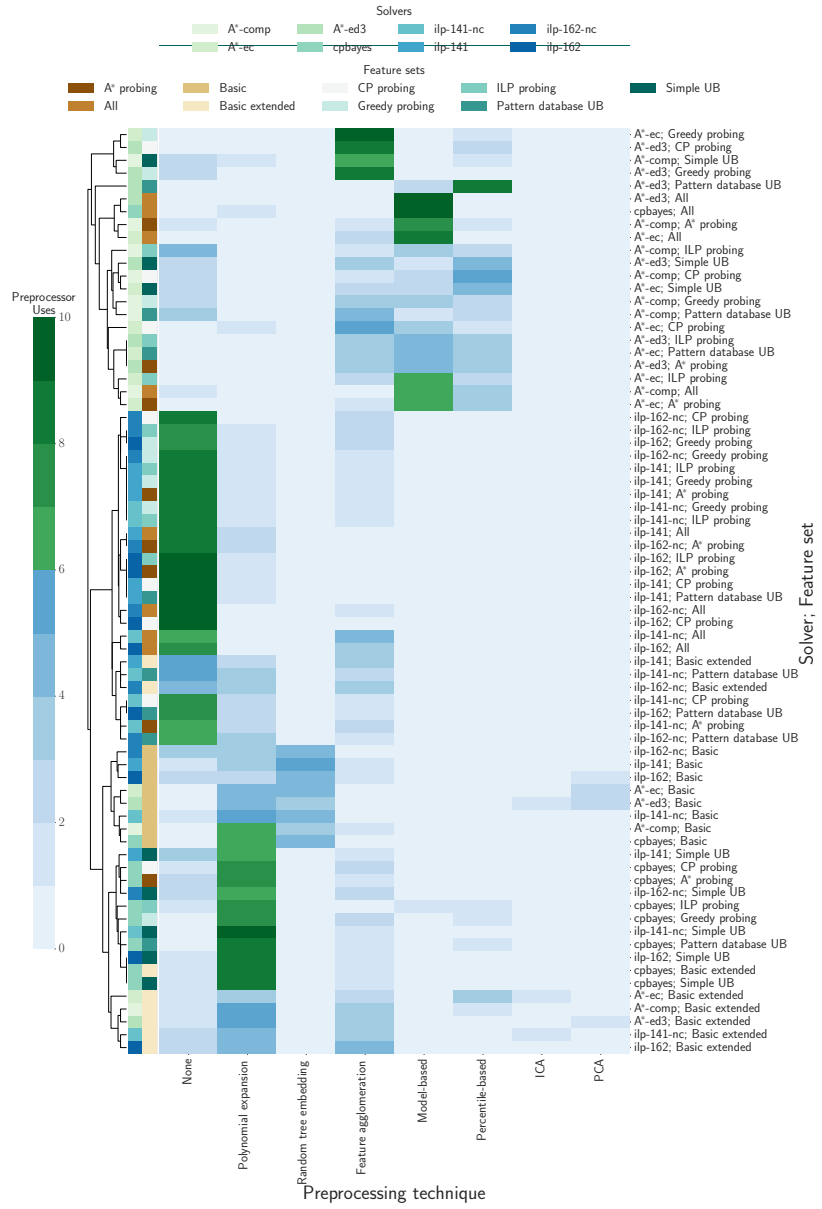


Fig. 11 The preprocessing techniques used by auto-sklearn for each combination of solver and feature set when learning a single random forest and preprocessor. The blue-green column of colors on the left indicate the solver in that row, and the green-brown column indicates the feature set; the text on the right also gives this information. Each cell shows the number of times the respective preprocessing technique was selected in one of the 10 cross-validation folds for the associated (solver, feature set) pair. The UPGMA algorithm [62] with a Euclidean distance metric was used for clustering. Cells shaded in green indicate statistically significantly high choices ($p < 0.01$, one-sided binomial test comparing to a uniform distribution, Benjamini-Hochberg multiple test correction).

an implicit tradeoff in auto-sklearn between resources used for preprocessing and the model itself.

Also unexpectedly, the trees for ILP without the graph-based cutting plane routines (the "-nc" variants) are much larger than those using it with the **ILP probing** feature set. The ILP implementation used in probing *does* use the graph-based cutting plane routines. The learner uses preprocessing only sparingly in all of these cases, so it again appears that a more complex model is used to handle the noise in the features.

6.4 Important Features

Finally, we computed the Gini importance [9] of each feature for predicting each solver while using the appropriate **Probing** features. The importance for a particular feature is calculated using a standard two-step technique [9]. First, the feature is corrupted with noise to create a new dataset. Then, the new dataset is used for training and testing as usual. The normalized increase in error when using the noisy feature is taken as its importance. For the random forests, this procedure is performed for all trees in the forest. The feature importance is then the average across all trees. Finally, we average the feature importances across each cross-validation fold.

Figure 12(b) shows important features for the different solvers. Several of the importances are unsurprising; the number of variables in the dataset determines the size of the search space for A*, and that was the most important feature for all variants. Similarly, the size of the linear program solved by ILP is directly determined by the number of CPSs, and its most important features describe the CPSs. Likewise, the respective probing error bound features were typically very important for ILP and CP. This is sensible because these features indicate when a solver can quickly converge to a nearly-optimal solution.

The CP solver uses relaxations from both A* and ILP; it shares many important features with the A* variants. Surprisingly, though, it only has one important feature in common with ILP. CP implicitly uses the pattern database relaxation, and the pattern database node degree is indeed one of its most important features. This suggests that the learner successfully identifies relevant features for the solvers.

In contrast to ILP and CP, A*-comp is the only A* variant for which probing was an important feature. Coupled with its worse RMSE shown in Figure 10 when using probing, this suggests that the runtime characteristics of the anytime variant of A* are different enough from A*-comp that it adds significant noise to learning.

Another somewhat unexpected result concerning A* is that many **Simple UB** features are quite important. Previous experimental results [70] show that the pattern database bounding approach is much more informative *during* the A* search. However, the solvers construct their pattern databases differently than those used for extracting features, so the structural properties, such as the number of non-trivial SCCs, of the constructed graphs may not reflect the difficulty of the problem for the solver.

In general, the results presented in Figure 12(b) reveal that a small number of features were consistently important for any particular solver; this is in line with previous work [41,43]. Qualitatively, this implies that most of the trees were based

on the same small set of features. Developing a more in-depth understanding of these instance characteristics in light of solver performance is an important aspect of future work.

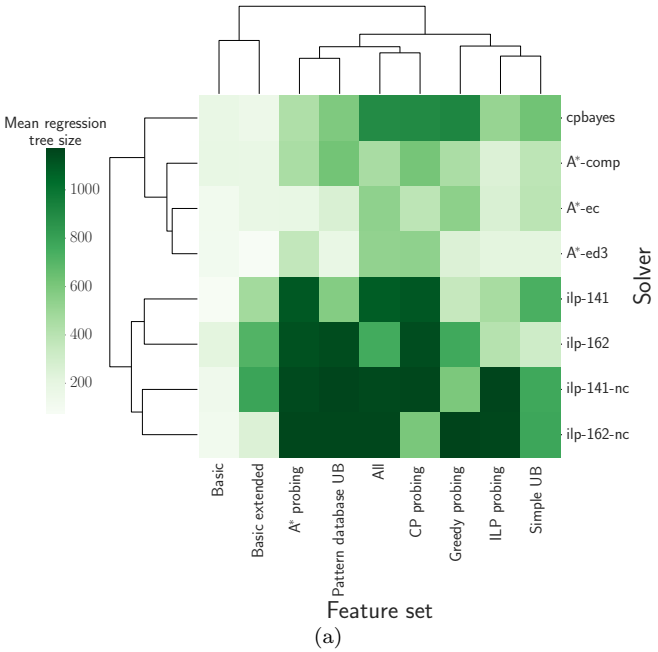
7 Conclusions

We have investigated the *empirical hardness* of BNSL, the Bayesian network structure learning problem, in relation to several state-of-the-art complete solvers, particularly solvers based on A* search, integer linear programming, and constraint programming. While each of these solvers always finds an optimal Bayesian network structure (w.r.t. a given scoring function), the running times of the solvers can vary greatly even within instances of the same size. Moreover, on a given instance, some solvers may run very fast, whereas others require considerably longer time, sometimes by several orders of magnitude. We validated this general view, which has emerged from a series of recent studies, by conducting the most elaborate evaluation of state-of-the-art solvers to date. We have made the rich evaluation data publicly available[12] in order to facilitate possible further analyses that go beyond the scope of the present work.

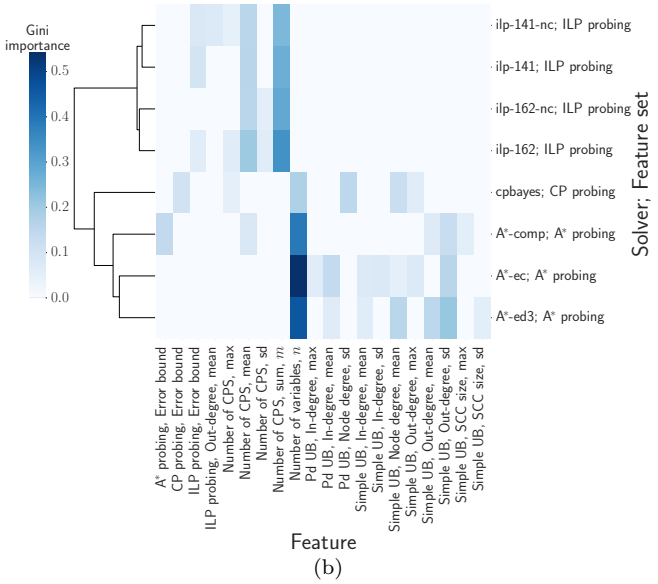
As the second contribution, we applied machine learning methods to construct *empirical hardness models* from the data obtained by the solver evaluations. Instantiating the general methodology of empirical hardness models (e.g., [56, 45]), we proposed several *features*, i.e., real-valued functions of BNSL instances, which are potentially informative about solver running times and which go beyond the basic parameters of instance size. As we cannot expect the features to capture all the variability in solver running times and we can only use limited training data, we adopted a statistical modeling approach. We used auto-sklearn [20], a state-of-the-art system for optimizing model class, preprocessing and relevant hyperparameters, for learning our regression models. For each solver and group of features, a dedicated empirical hardness model was learned and evaluated.

The learned models allowed us to answer two basic questions concerning our ability to predict the solvers' relative and absolute performance before actually running the solvers. The first question (Q1) asked whether the basic parameters of input size suffice for reliably predicting which of the solvers is the fastest on a given problem instance. We answered this question in the affirmative by showing that whenever a solver is significantly slower than the fastest solver on a given instance, the slower one is very rarely predicted as the fastest one. Naturally, when the differences between the fastest solvers are small, predicting which one is the fastest among them gets less reliable, but also less important. Indeed, we demonstrated the utility of the learned empirical hardness model by building an efficient solver portfolio, which exploits the high running time variance over the different solvers. For varying distributions of instances, our portfolio solver runs nearly as fast as the fastest solver overall. In contrast, the cumulative running time of the best individual solver is over five times that of the VBS. As a result, the proposed solver portfolio is currently the fastest algorithm for solving BNSL when averaged over a large heterogeneous set of instances.

[12] `http://bnportfolio.cs.helsinki.fi/`



(a)



(b)

Fig. 12 (a) The average size of the regression trees in the random forests learned by auto-sklearn for each solver and feature set. (b) The Gini importance [9] of features in the learned random forest models for each solver using the respective **Probing** feature set. Only features with an importance of at least 0.05 for at least one solver are included. We use the abbreviations "Pd" for pattern database and "sd" for standard deviation. The UPGMA algorithm [62] with a Euclidean distance metric was used for clustering in both cases; the features in (b) were not clustered.

Our answer was affirmative also to the second question (Q2) of whether the running times of each of the solvers can be predicted significantly more accurately by extending the set of features. We observed that, in general, the more high-quality features, the more accurate predictions. In the solver portfolio performance, however, the more accurate running time predictions translated only to a small improvement. This was somewhat expected since the simple portfolio already achieved very good performance.

Via the extensive empirical evaluation presented as part of this work, we managed to answer some of the key basic questions about the empirical hardness of BNSL. This first study opens several avenues for future research. First, we believe the proposed collection of features is not complete—presumably, there are even more informative, albeit possibly slower-to-compute, features yet to be discovered. The question of how to efficiently trade informativeness for computational efficiency is relevant also more generally for the algorithm selection methodology; probing features [33], as applied in this work to the context of BNSL, provide just one, rather generic technique. Second, the empirical hardness model and its evaluated performance obviously depend on the distribution of the training and test instances. While this dependency is unavoidable, it is an intriguing question to what extent the dependency can be weakened by considering appropriate distributions and sufficiently large samples of instances. In the course of answering these questions, we also validated that the models learned by auto-sklearn capture many sensible characteristics of the solvers.

Finally, we note that while in this work we focused on understanding and predicting the runtime behavior of complete BNSL solvers, i.e., exact algorithms that provide provably-optimal solutions to given BNSL instances, the techniques studied and developed in this paper could also be extended to cover in-exact local-search style, greedy, and approximate algorithmic approaches to BNSL. While such approaches exhibit typically better scalability than the here-studied exact approaches, the fact that in-exact approaches cannot give guarantees of optimality on the produced solutions brings new challenges in terms of portfolio construction and prediction, specifically in understanding the interplay between solution quality and runtimes. Another potentially interesting direction—although somewhat secondary aspect compared to runtime behavior—for further study would be to understand and predict the memory usage of exact approaches. Furthermore, it would be interesting expand the study in the future by including additional datasets e.g. from OpenML [66]

Acknowledgements The authors thank James Cussens for discussions on GOBNILP and the anonymous reviewers for valuable suggestions that helped improve the manuscript.

References

1. Achterberg, T.: SCIP: Solving constraint integer programs. Mathematical Programming Computation **1**(1), 1–41 (2009)
2. Bache, K., Lichman, M.: UCI machine learning repository (2013). URL `http://archive.ics.uci.edu/ml`
3. Balint, A., Belov, A., Järvisalo, M., Sinz, C.: Overview and analysis of the SAT Challenge 2012 solver competition. Artificial Intelligence **223**, 120–155 (2015)
4. Bartlett, M., Cussens, J.: Integer linear programming for the Bayesian network structure learning problem. Artificial Intelligence **In press** (2015)

5. van Beek, P., Hoffmann, H.: Machine learning of Bayesian networks using constraint programming. In: Proceedings of the 21st International Conference on Principles and Practice of Constraint Programming (CP 2015), *Lecture Notes in Computer Science*, vol. 9255, pp. 429–445. Springer (2015)
6. Berg, J., Järvisalo, M., Malone, B.: Learning optimal bounded treewidth Bayesian networks via maximum satisfiability. In: Proceedings of the 17th International Conference on Artificial Intelligence and Statistics (AISTATS 2014), *JMLR Workshop and Conference Proceedings*, vol. 33, pp. 86–95. JMLR (2014)
7. Bielza, C., Larrañaga, P.: Discrete bayesian network classifiers: A survey. ACM Comput. Surv. **47**(1), 5:1–5:43 (2014)
8. Bischl, B., Kerschke, P., Kotthoff, L., Lindauer, M.T., Malitsky, Y., Fréchette, A., Hoos, H.H., Hutter, F., Leyton-Brown, K., Tierney, K., Vanschoren, J.: ASlib: A benchmark library for algorithm selection. Artificial Intelligence **237**, 41–58 (2016). DOI 10.1016/j.artint.2016.04.003. URL `http://dx.doi.org/10.1016/j.artint.2016.04.003`
9. Breiman, L.: Random forests. Machine Learning **45**, 5–32 (2001)
10. Buntine, W.: Theory refinement on Bayesian networks. In: Proceedings of the 7th Conference on Uncertainty in Artificial Intelligence (UAI 1997), pp. 52–60. Morgan Kaufmann Publishers Inc. (1991)
11. de Campos, C., Ji, Q.: Efficient learning of Bayesian networks using constraints. Journal of Machine Learning Research **12**, 663–689 (2011)
12. Carbonell, J., Etzioni, O., Gil, Y., Joseph, R., Knoblock, C., Minton, S., Veloso, M.: Prodigy: an integrated architecture for planning and learning. SIGART Bulletin **2**, 51–55 (1991)
13. Cheng, J., Greiner, R., Kelly, J., Bell, D.A., Liu, W.: Learning Bayesian networks from data: An information-theory based approach. Artificial Intelligence **137**(1-2), 43–90 (2002)
14. Chickering, D.: Learning Bayesian networks is NP-complete. In: Learning from Data: Artificial Intelligence and Statistics V, pp. 121–130. Springer-Verlag (1996)
15. Cooper, G., Herskovits, E.: A Bayesian method for the induction of probabilistic networks from data. Machine Learning **9**, 309–347 (1992)
16. Cussens, J.: Bayesian network learning with cutting planes. In: Proceedings of the 27th Conference on Uncertainty in Artificial Intelligence (UAI 2011), pp. 153–160. AUAI Press (2011)
17. Cussens, J.: Advances in Bayesian network learning using integer programming. In: Proceedings of the 29th Conference on Uncertainty in Artificial Intelligence (UAI 2013), pp. 182–191. AUAI Press (2013)
18. Fan, X., Malone, B., Yuan, C.: Finding optimal Bayesian network structures with constraints learned from data. In: Proceedings of the 30th Conference on Uncertainty in Artificial Intelligence (UAI 2014), pp. 200–209. AUAI Press (2014)
19. Fan, X., Yuan, C.: An improved lower bound for Bayesian network structure learning. In: Proceedings of the 29th AAAI Conference on Artificial Intelligence (AAAI 2015), pp. 3526–3532. AAAI Press (2015)
20. Feurer, M., Klein, A., Eggensperger, K., Springenberg, J., Blum, M., Hutter, F.: Efficient and robust automated machine learning. In: Advances in Neural Information Processing Systems 28 (2015)
21. Fink, E.: How to solve it automatically: Selection among problem-solving methods. In: Proceedings of the 4th International Conference on Artificial Intelligence Planning Systems (AIPS 1998), pp. 126–136. AAAI Press (1998)
22. Fréchette, A., Kotthoff, L., Michalak, T.P., Rahwan, T., Hoos, H.H., Leyton-Brown, K.: Using the shapley value to analyze algorithm portfolios. In: D. Schuurmans, M.P. Wellman (eds.) Proceedings of the 30th AAAI Conference on Artificial Intelligence, pp. 3397–3403. AAAI Press (2016)
23. Friedman, N., Koller, D.: Being Bayesian about network structure. a Bayesian approach to structure discovery in Bayesian networks. Machine Learning **50**, 95–125 (2003)
24. Gebruers, C., Hnich, B., Bridge, D.G., Freuder, E.C.: Using CBR to select solution strategies in constraint programming. In: 6th International Conference on Case-Based Reasoning (ICCBR 2005), *Lecture Notes in Computer Science*, vol. 3620, pp. 222–236. Springer (2005)
25. Giraud-Carrier, C., Vilalta, R., Brazdil, P.: Introduction to the special issue on meta-learning. Machine Learning **54**(3), 187–193 (2004)
26. Gomes, C.P., Selman, B.: Algorithm portfolios. Artificial Intelligence **126**(1-2), 43–62 (2001)

27. Hall, M., Frank, E., Holmes, G., Pfahringer, B., Reutemann, P., Witten, I.H.: The WEKA data mining software: an update. SIGKDD Explorations **11**(1), 10–18 (2009)
28. Heckerman, D., Geiger, D., Chickering, D.: Learning Bayesian networks: The combination of knowledge and statistical data. Machine Learning **20**, 197–243 (1995)
29. Hoos, H., Lindauer, M.T., Schaub, T.: claspfolio 2: Advances in algorithm selection for answer set programming. Theory and Practice of Logic Programming **14**(4-5), 569–585 (2014)
30. Horvitz, E., Ruan, Y., Gomes, C.P., Kautz, H.A., Selman, B., Chickering, D.M.: A Bayesian approach to tackling hard computational problems. In: Proceedings of the 17th Conference on Uncertainty in Artificial Intelligence (UAI 2001), pp. 235–244. Morgan Kaufmann (2001)
31. Hurley, B., Kotthoff, L., Malitsky, Y., O'Sullivan, B.: Proteus: A hierarchical portfolio of solvers and transformations. In: Proceedings of the 11th International Conference on Integration of AI and OR Techniques in Constraint Programming (CPAIOR 2014), *Lecture Notes in Computer Science*, vol. 8451, pp. 301–317. Springer (2014)
32. Hutter, F., Hoos, H.H., Leyton-Brown, K.: Sequential model-based optimization for general algorithm configuration. In: Selected Papers of the 5th International Conference on Learning and Intelligent Optimization (LION 5), *Lecture Notes in Computer Science*, vol. 6683, pp. 507–523. Springer (2011)
33. Hutter, F., Xu, L., Hoos, H.H., Leyton-Brown, K.: Algorithm runtime prediction: Methods & evaluation. Artificial Intelligence **206**, 79–111 (2014)
34. Jaakkola, T.S., Sontag, D., Globerson, A., Meila, M.: Learning Bayesian network structure using LP relaxations. In: Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics (AISTATS 2010), *JMLR Proceedings*, vol. 9, pp. 358–365. JMLR.org (2010)
35. Järvisalo, M., Le Berre, D., Roussel, O., Simon, L.: The international SAT solver competitions. AI Magazine **33**(1), 89–92 (2012)
36. Koivisto, M., Sood, K.: Exact Bayesian structure discovery in Bayesian networks. Journal of Machine Learning Research pp. 549–573 (2004)
37. Kontkanen, P., Myllymäki, P.: MDL histogram density estimation. In: In Proceedings of the Eleventh International Conference on Artificial Intelligence and Statistics (AISTATS 2007), *JMLR Proceedings*, vol. 2, pp. 219–226. JMLR.org (2007)
38. Kotthoff, L.: Algorithm selection for combinatorial search problems: A survey. AI Magazine **35**(3), 48–60 (2014)
39. Kotthoff, L., Gent, I.P., Miguel, I.: An evaluation of machine learning in algorithm selection for search problems. AI Communications **25**(3), 257–270 (2012)
40. Kotthoff, L., Kerschke, P., Hoos, H., Trautmann, H.: Improving the state of the art in inexact TSP solving using per-instance algorithm selection. In: Revised Selected Papers of the 9th International Conference on Learning and Intelligent Optimization (LION 9), *Lecture Notes in Computer Science*, vol. 8994, pp. 202–217. Springer (2015)
41. Lee, J.W., Giraud-Carrier, C.G.: Predicting algorithm accuracy with a small set of effective meta-features. In: Proceedings of the 7th International Conference on Machine Learning and Applications (IEEE ICMLA 2008), pp. 808–812. IEEE Computer Society (2008)
42. Leite, R., Brazdil, P., Vanschoren, J.: Selecting classification algorithms with active testing. In: Proceedings of the 8th International Conference on Machine Learning and Data Mining in Pattern Recognition (MLDM 2012), *Lecture Notes in Computer Science*, vol. 7376, pp. 117–131. Springer (2012)
43. Leyton-Brown, K., Hoos, H.H., Hutter, F., Xu, L.: Understanding the empirical hardness of *NP*-complete problems. Commun. ACM **57**(5), 98–107 (2014)
44. Leyton-Brown, K., Nudelman, E., Shoham, Y.: Learning the empirical hardness of optimization problems: The case of combinatorial auctions. In: 8th International Conference on Principles and Practice of Constraint Programming (CP 2002), *Lecture Notes in Computer Science*, vol. 2470, pp. 556–572. Springer (2002)
45. Leyton-Brown, K., Nudelman, E., Shoham, Y.: Empirical hardness models: Methodology and a case study on combinatorial auctions. Journal of the ACM **56**(4) (2009)
46. Lobjois, L., Lemaître, M.: Branch and bound algorithm selection by performance prediction. In: Proceedings of the 15th National Conference on Artificial Intelligence (AAAI 1998), pp. 353–358. AAAI Press (1998)
47. Madigan, D., York, J.: Bayesian graphical models for discrete data. International Statistical Review **63**, 215–232 (1995)

48. Malone, B., Järvisalo, M., Myllymäki, P.: Impact of learning strategies on the quality of Bayesian networks: An empirical evaluation. In: Proceedings of the 31st Conference on Uncertainty in Artificial Intelligence (UAI 2015), pp. 362–371. AUAI Press (2015)
49. Malone, B., Kangas, K., Järvisalo, M., Koivisto, M., Myllymäki, P.: Predicting the hardness of learning Bayesian networks. In: Proceedings of the 28th AAAI Conference on Artificial Intelligence (AAAI 2014), pp. 2460–2466. AAAI Press (2014)
50. Malone, B.M., Yuan, C.: Evaluating anytime algorithms for learning optimal Bayesian networks. In: Proceedings of the 29th Conference on Uncertainty in Artificial Intelligence (UAI 2013). AUAI Press (2013)
51. Ott, S., Imoto, S., Miyano, S.: Finding optimal models for small gene networks. In: Proceedings of the Pacific Symposium on Biocomputing 2004, pp. 557–567. World Scientific (2004)
52. Parviainen, P., Koivisto, M.: Finding optimal Bayesian networks using precedence constraints. Journal of Machine Learning Research **14**, 1387–1415 (2013)
53. Pearl, J.: Probabilistic reasoning in intelligent systems: networks of plausible inference. Morgan Kaufmann (1988)
54. Perrier, E., Imoto, S., Miyano, S.: Finding optimal Bayesian network given a superstructure. Journal of Machine Learning Research **9**, 2251–2286 (2008)
55. Pulina, L., Tacchella, A.: Treewidth: A useful marker of empirical hardness in quantified Boolean logic encodings. In: Proceedings of the 15th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning (LPAR 2008), *Lecture Notes in Computer Science*, vol. 5330, pp. 528–542. Springer (2008)
56. Rice, J.: The algorithm selection problem. Advances in Computers **15**, 65–118 (1976)
57. Rijn, J.N., Abdulrahman, S.M., Brazdil, P., Vanschoren, J.: Fast algorithm selection using learning curves. In: Proceedings of the 14th International Symposium on Advances in Intelligent Data Analysis (IDA 2015), *Lecture Notes in Computer Science*, vol. 9385, pp. 298–309. Springer (2015)
58. Saikko, P., Malone, B., Järvisalo, M.: MaxSAT-based cutting planes for learning graphical models. In: Proceedings of the 12th International Conference on Integration of Artificial Intelligence and Operations Research Techniques in Constraint Programming (CPAIOR 2015), *Lecture Notes in Computer Science*, vol. 9075, pp. 345–354. Springer (2015)
59. Shapley, L.S.: A value for n-person games. Contributions to the theory of games **2**, 307–317 (1953)
60. Silander, T., Myllymäki, P.: A simple approach for finding the globally optimal Bayesian network structure. In: Proceedings of the 22nd Conference in Uncertainty in Artificial Intelligence (UAI 2006), pp. 445–452. AUAI Press (2006)
61. Singh, A., Moore, A.: Finding optimal Bayesian networks by dynamic programming. Tech. rep., Carnegie Mellon University (2005)
62. Sokal, R.R., Michener, C.D.: A statistical method for evaluating systematic relationships. The University of Kansas Science Bulletin **38**(2), 1409–1438 (1958)
63. Spirtes, P., Glymour, C., Schemes, R.: Causation, Prediction, and Search. Springer, New York (1993)
64. Tamada, Y., Imoto, S., Miyano, S.: Parallel algorithm for learning optimal Bayesian network structure. Journal of Machine Learning Research **12**, 2437–2459 (2011)
65. Teyssier, M., Koller, D.: Ordering-based search: A simple and effective algorithm for learning Bayesian networks. In: Proceedings of the 21st Conference in Uncertainty in Artificial Intelligence (UAI 2005), pp. 584–590. AUAI Press (2005)
66. Vanschoren, J., van Rijn, J.N., Bischl, B., Torgo, L.: OpenML: networked science in machine learning. SIGKDD Explorations **15**(2), 49–60 (2013)
67. Wunderling, R.: Paralleler und objektorientierter Simplex-Algorithmus. Ph.D. thesis, Technische Universität Berlin (1996)
68. Xu, L., Hutter, F., Hoos, H., Leyton-Brown, K.: SATzilla: Portfolio-based algorithm selection for SAT. Journal of Artificial Intelligence Research **32**, 565–606 (2008)
69. Yuan, C., Malone, B.: An improved admissible heuristic for finding optimal Bayesian networks. In: Proceedings of the 27th Conference in Uncertainty in Artificial Intelligence (UAI 2012), pp. 924–933. AUAI Press (2012)
70. Yuan, C., Malone, B.: Learning optimal Bayesian networks: A shortest path perspective. Journal of Artificial Intelligence Research **48**, 23–65 (2013)

Appendix A Details on the Data Sets

The numbers of variables and records in each of the data sets used in the experiments are shown in Table 7 (for Real) and in Table 8 (for Sampled).

Table 7 Sizes of the datasets in Real.

Dataset	#Variables	#Records
letter	17	20000
voting	17	435
zoo	17	101
lymph	19	148
eucalyptus	20	736
hepatitis	20	155
credit-g	21	1000
hypothyroid	22	3772
mushroom	22	8124
spect	23	267
autos	26	205
colic	28	368
pyrim	28	74
flag	29	194
trains	30	10
anneal	32	898
backache	32	180
marketing	33	364
student-mat	33	395
student-por	33	649
turkiye	33	5820
dermatology	35	366
soybean	36	307
kr-vs-kp	37	3196
stemmatology	37	1208
abscisic	41	5456
diabetes	41	60000
connect-4_6000	43	6000
connect-4_60000	43	60000
covtype_60000	43	60000
sponge	45	76
wiki4he	53	913
lung-cancer	57	32
promoters	58	106
triazines	59	186
splice	61	3190
audiology_63	63	226
optdigits	63	5620
plants_63	63	34781

Table 8 Sizes of the datasets in SAMPLED.

Dataset	**#Variables**	**#Records**
kredit	18	1000
insurance	27	100; 1000; 10,000
water	32	100; 1000; 10,000
mildew	35	100; 1000; 10,000
alarm	37	100; 1000; 10,000
hailfinder	56	100; 1000; 10,000
carpo	60	100; 1000; 10,000